Building a Smart High Quality Software Pipeline

Author(s)

Richard.Robinson@bongolearn.com

Abstract

How can we ensure quality with quick releases full of great features and changes in today's modern software world?

As software development, integration, and delivery processes continue to speed up with increased demand and growth throughout the industry there is a growing need to evolve our quality assurance approaches to ensure solid and quality code arrives in the hands of every user along with a constant flow of new features and changes. Quick releases, stringent SLAs for uptime and service, massive and quick scale up/down needs, accessibility, localization, and a huge array of user devices are some of the challenges that can cause significant issues internally and for our customers without the right changes.

We are constantly evolving our own approach and have learned some good lessons along the way on how to build in quality at each phase with automation and integration of tools and processes throughout our CI/CD pipeline to significantly decrease our hotfixes and interruptions in production while increasing our delivery of features to our customer base. We have gone from significant down time each release in each supported region worldwide to zero downtime releases with better automated and manual verification tests along the way improving efficiencies of our engineers internally and our customers in production.

We're still learning, but in this paper, I share some of the lessons from our journey that could help others with practical strategies and approaches to enable quality software in a modern CI/CD pipeline world.

Biography

Richard Robinson currently leads the DevOps, QA, Infrastructure, and IT engineering groups supporting all operations for BongoLearn, Inc. We at Bongo create video training, learning, and assessment workflows and technologies embedded in many of the prominent Learning Management Systems in the EdTech industry. I have a background in QA, automation, and managing QA and systems engineering teams for large and small companies over the last 22 years. I am passionate about quality and ensuring customers and companies can effectively use technology to solve their problems without needing to worry about issues and problems that plague our world and get in the way of real work.

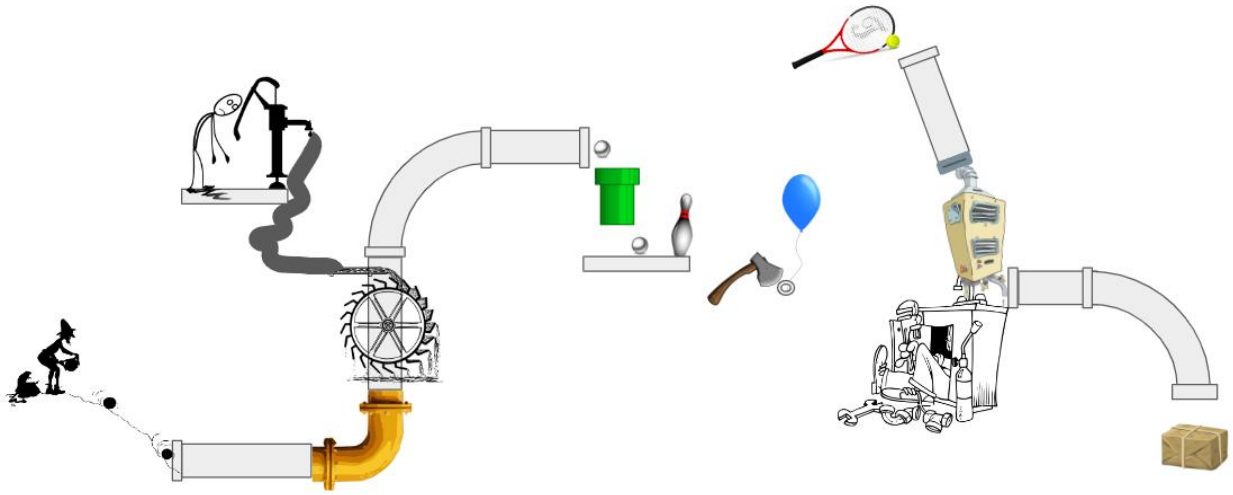
1 Introduction

Much of the modern software engineering world is transforming from larger releases to smaller, more frequent releases. So how do we support integration and release pipelines that are tailored to these types of releases? We might start with just a series of different phases of the software development lifecycle loosely connected like a Rube Goldberg machine sometimes getting stuck and needing a nudge to go to the next step. Then with the right planning and perseverance this can evolve into a robust, dependable, and fully automated smart pipeline enabling continuous integration of new/modified code and quality delivery of that to production to delight end users.

The answer I think lies in a lot of the contributing aspects of software development partnered with operations and modern DevOps principles applied in a quality fashion. It is also constantly evolving and will look different for different businesses and domains. Critical software that could kill someone might use different approaches or levels than less critical software, but I believe there is a huge overlap in the principles, approaches, and tools that can be utilized. This is similar to differences between 99.999% uptime and 99.9% uptime for different software for different purposes seeking "…to balance the risk of unavailability with the goals of rapid innovation and efficient service operations, so that users' overall happiness—with features, service, and performance—is optimized." The risk tolerance of failures compared to engineering cost is continually considered and can even be measured like Google's SRE (Site Reliability Engineering) Error Budgets.

So much depends on what kind of pipeline we are building and all the contributing factors to what goes into that pipeline. From the right tools for good development environments for individual software engineers to deployment tools and processes ensuring solid delivery and execution of code in a variety of real-world environments. In the middle we find things like a good code review process and culture, code analysis tools, maintainable automated test suites, Infrastructure as Code (IaC), and other positively contributing factors.

I will cover some of these areas and what we have found most helpful for evolving from a rusty old pipe, or sections of pipe, loosely held together to get software into the hands of our end users to a pipe that while not finished (are CI/CD pipelines ever finished?) has dramatically improved our ability to deliver quality software applications to our customers.



2 Enabling Zero Downtime Releases

Four-hour downtime maintenance windows to zero downtime releases

A few years ago, it was typical for our customers to experience three to four hours of downtime showing a construction page or an unhelpful error for each maintenance window. With six production environments deployed throughout the world that was becoming pretty noticeable and painful to our end customers, partners, and the engineering team manually deploying all of those environments (4 hours x 6 environments = 24 hours for each release). Those three to four hours were often set from midnight to 3 or 4 am to minimize impact on end users which further negatively impacted the engineering team involved in each release. Increasing the release frequency with such a situation was clearly untenable and human errors were common.

To move from this situation took a few key initial first steps:

- First, we needed to understand all the pieces and each step involved in the deployment.
- Second, we created a checklist to ensure that the manual and brittle process could be performed without missing key steps to keep supporting the business by releasing new features and fixes until automation could be built.
- Third, we needed to investigate and employ techniques and tools so each step could be performed without bringing the system down.
- Fourth (although really started after the first step), we needed representative internal environments to test the whole process and automate against.

The first and second steps go nicely together as we documented which steps were being done and which needed to happen in what order reviewing and discussing as we went. This laid out almost a blueprint of sorts for all the following steps and the evolution as we prioritized which steps to optimize and automate later. The checklists we then used to execute the deployment of a release didn't actually need every detail and specific piece, but rather reminders of the key steps, correct sequencing, and overall flow. The pipeline started to take shape and we even saw the satisfaction as we turned the checklist items that were automated a different color while retaining them in the checklist. Atul Gawande in his book "The Checklist Manifesto: How to Get Things Right" reinforces the point that in today's modern world of complexity whether in surgery (he personally is a surgeon), building modern skyscrapers, flying an airplane, or building and deploying software the volume and complexity our human brains are dealing with needs a level of assistance to consistently get it right. "…the volume and complexity of what we know has exceeded our individual ability to deliver its benefits correctly, safely, or reliably."

The third step we took was to investigate all of the approaches for delivering software without any impact to end users or need for a customer visible maintenance window. This took time to shift to use approaches like blue/green or black/red approaches. Basically, most of the approaches deal with delivering a set of software alongside the current running software and flipping it over to run with the newer version either all at once or incrementally as multiple nodes within a cluster of nodes are updated and users start hitting the new code. This can often be partnered with feature flags so the new code can be in place with the same behavior as the old code until a flag is triggered to activate the new code. These and various other approaches were key to shifting the mindset so there was no impact to an end user until they just clicked on the next page/button/menu or the next API call and started using new code. As is probably the case with most software projects we had to take each part and incrementally make changes to accommodate that and adopt new software practices to maintain backward compatibility and ensure each new release could also be deployed without impact to end users. Significant testing and verification were performed on the application as a whole and on the specific areas of change to ensure changes could be rolled out at any time. Also, I'll admit that to begin with until we were more confident in our processes, we still deployed in off hours in case something was missed to minimize potential impact and now we regularly release any time and even at usage peaks in the middle of the day.

The fourth and either final or ongoing step after identifying what steps need to be taken is to ensure that there are representative internal environments that mimic production so well that the whole process can be created and verified outside of production. This will likely require several iterations to work out the differences between internal environments and actual production environments and ongoing work to keep the environments 'in sync'. The industry typically has the concept of staging environments where new software can be staged before deploying to production or switching between production and staging and back again as blue goes to green and then back again to blue. In whatever way it is implemented the important principle I think is to make sure there are internal environments that are close enough to production that steps, checklists, and automation used against those environments will perform the same as in production. This should be considered all the way back to development environments for engineers as much as possible.

A key item to capture and include through all these steps is adequate testing and verification steps whether performed manually to begin with or in an automated fashion. These are key to ensuring quality as this process evolves and in the end result. Wherever an organization is in this evolution I believe implementing even some of these steps will start to yield the desired results and the return on investment of these activities can fuel the engineering necessary to continue the evolutionary process. While it would be nice to pause everything and create the whole process and then make it live I believe most organizations are in the position where incremental progress evolving something that already exists is what reasonably can be done. Of course, if a brand-new process is being constructed for a brand-new application, I think these same steps can still be used enabling an orderly design and creation. Like Test Driven Development (TDD) principles where tests are created first which all fail until the code is written that enables the tests to pass these steps should provide a nice framework for even a green field CI/CD pipeline for zero downtime deployments.

3 Building Confidence in Test Suites

How to build confidence in your manual and automated regression suites

The quality of the testing, validation, and verification capability of a software development organization is key to building a pipeline you will be able to use and rely on and especially as was just mentioned with zero downtime deployments at any time of day.

Some initial pieces to start with:

- Internal environments and a pure CI (Continuous Integration) target where each check-in of code is put into an environment with other code and tests are run.
- Some of the most basic tests just to build out the process and build that "new muscle". Even a smoke test run against a build of some new code put into even the most basic integration environment can be a great foundation to start with.
- Manual test suites alongside any sort of automated unit tests, functional tests, regression suites, or other more specialized tests so there is always a clear view of what is being covered and what could be the next priority to build up a solid automated suite.
- Using test suites regularly for internal environments and then injected into the correct places in the manual, partially automated, or fully automated deployment pipeline.

As mentioned earlier, using a checklist approach of specific items to cover in the correct sequence and configuration helps ensure quality in the short-term and a map of what needs to be created to take humans out of the execution phase. As we take humans, and our human engineering hours, out of the execution phase and more into the design and architecting of test approaches, deployment considerations, etc. quality improves, and the overall engineering process can accelerate without compromising quality. As the checklists and results are reviewed the checklist evolves to include the items that are missed in previous iterations and there is a clearer map of the items that can and should be

automated to put the pieces of the pipeline together. One powerful interim result here is that even without all the pieces automated together benefits are still realized as each piece itself is solidified and improved.

Risk-based testing approaches can be used to ensure the evolution of the pipeline is providing Return on Investment (ROI) as we go, and we are hitting on the most critical items to the business or the current user base. Whenever tests are considered for a given application or feature it can be helpful to go through a list of potential dimensions or considerations to determine which are more, or less, significant, or unique to that particular application, use case, or domain. Considering all the various dimensions up front enables a team to consciously and more holistically prioritize or deprioritize and not just prioritize what is currently in the list or top of mind and possibly miss some area that might be even more significant.

Of course, areas of functionality for the product should also be considered and where changes are being made in a particular release that might impact one area more than another area and warrant additional testing. This is especially important when that testing in the short-term requires manual test effort. Knowing whether to do a light touch of an area or a deeper regression test is important to budgeting QA time and enable the appropriate build out of additional automated tests and complex testing scenarios/configs.

Some potential areas to consider:

High-level Area or Consideration	**Details/Description**
Accessibility	Does this system properly support accessibility with low vision and screenreader support? Also closed captioning capabilities
Performance	Performance tests to ensure that the system's response times and throughput meet the user expectations and meet specified performance criteria or goals.
Scale/Sizing	Horizontal and Vertical scaling considerations
Stress	Pushing the boundaries of performance limits or even just limits of specific fields/types (e.g., numeric limits for integer fields or overflowing string sizes especially when storing in DB tables with columns of certain types)
Security & Privacy	Security tests to determine how secure the system is and if specific security requirements have been met. Could include GDPR, FERPA, HIPAA, data residency restrictions, etc.
Upgrade and Migration	Data preserved after upgrades and properly migrated to any new formats
Stability/Reliability	Tests performed to run the product in a customer-like environment over a period of time to verify that the system remains stable and there are no significant memory/handle/thread leaks or degradation to the system over time.
Usability	Is the system usable without intensive training or use of workarounds? Is it suitable for the target user community? Ease of use and standard UI behavior is good to consider here as well including use of phones, tablets, laptops, etc.
Supportability and Maintainability	Are there logs, debug levels, design docs, troubleshooting guides, API specs, and other things in place so that the product can be supported
Etc.	

I have found that reviewing a list of these types of considerations and discussing with others on the team enables reasonable tradeoffs to be made and appropriate plans to automate, adjust the pipeline, and otherwise improve the process without compromising quality or generating unacceptable risk.

In order to keep momentum during the evolution of an automated test suite and not lose ground it is important to include as part of the standard done criteria the automated deployment capability as well as automated unit, integration, end to end, and other tests. There are a lot of forces that can work against that (time pressure, additional engineering cost, etc.). Practical tradeoffs/compromises sometimes needed to keep overall solid strategy moving forward balancing business and market needs and high-quality engineering needs. However, even if some short-term tradeoffs need to be made the overall strategy can be protected as current automated verification capability is protected as a key priority along with properly working features.

For example: A new feature X is needed to be delivered as quickly as possible and impacts current functionality/behavior. At a minimum the current automated suite needs to be preserved by making it compatible with changes for the new feature even if full testing around new functionality needs to be manually finished first and then a follow-on sprint after release remaining automated tests fleshed out to ensure ongoing automated coverage. Designing a test suite for maintainability and to expect changes is of course a significant advantage for the current and continuing evolving application and pipeline.

One strategy that can assist with these short-term needs and incremental evolution is to build in processes to allow the pipeline to support 'augmentation' as well as 'fully automated' pieces. We have found it helpful to have automated scripts which sometimes only partially automate complex steps or even have pauses to enable a human to interject a manual step into the process so the overall process can flow with the nudges to the Rube Goldberg machine necessary in the short-term knowing that those will be eliminated over time with fully automated and robust processes in the future. There are some good approaches in the industry to even have scripts or automated pieces that are just placeholders for a manual step until the placeholder can be 'fleshed out' with actual automation that performs the task. Of course, with all of these processes good software engineering principles should be adhered to so code is re-used properly, properly checked in, managed, and reviewed by others on the team and regularly tested as a part of the overall solution.

4 Components of a Solid Pipeline

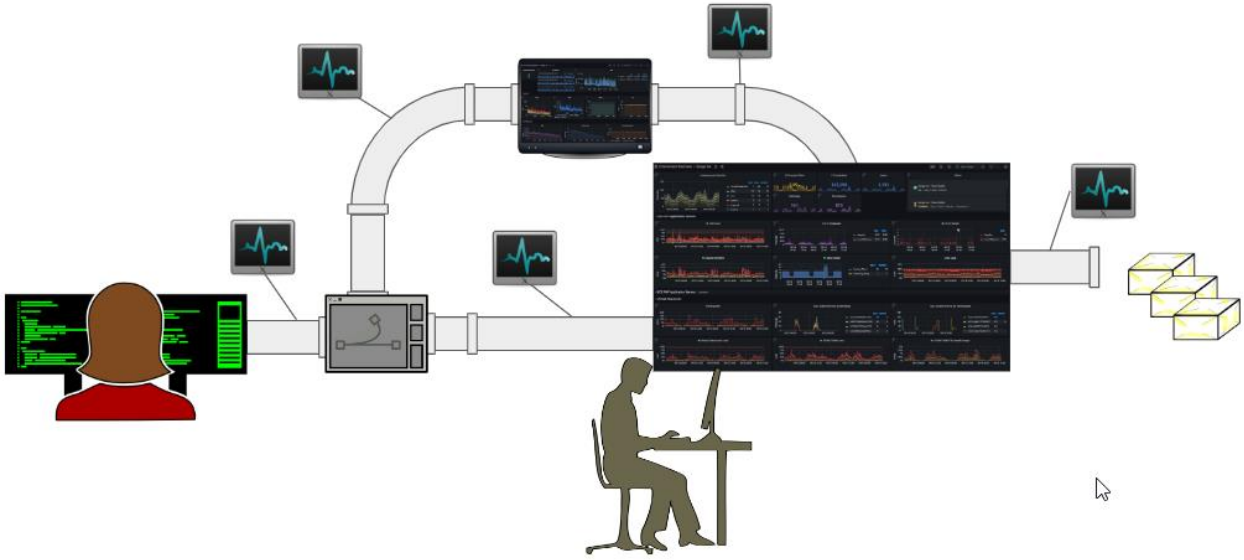
Important layers and aspects of a solid CI/CD pipeline

Some of the important layers that need to be formed around a CI/CD pipeline to enable the pipe to be solid now and in the future are:

- Flexibility - need to be able to fit in a variety of tools and processes. Even manual processes that might need to be interjected in.
- Code scanning - static code analysis for quality, security scans, licensing, etc.
- Test execution – unit tests, functional tests, integration tests, more end-to-end tests, and even specialty tests like perf/load/scale or accessibility tests can be added in
- Monitoring of behavior as software goes through the pipeline and in production. If there is no visibility into the pipe until it comes out the other end, then that just adds risk and surprises into a system that needs to be robust and deterministic.
- Tools like ELK (ElasticSearch, Logstash, and Kibana), Sentry, or others to enable thorough monitoring while things are in internal environments, traveling through the pipeline, and in production

- Incorporating automated functional tests to explore load, scale, and performance as well as to enhance monitoring with synthetic usage (good for regular heartbeat monitoring, exact/expected results in production alongside real-world usage)

As instrumentation is added into a pipeline and into production environments then the overall process becomes much more solid and reliable, and everyone benefits. It will also be easier to spot the bottlenecks or where issues are most prevalent for proper prioritization of next steps.



5 Summary/Conclusion

In summary I would like to review a few of the key principles that can be applied regardless of business, domain, or toolset to help with your building or enhancing of a smart, high quality, software pipeline.

- Checklists - ensure immediate quality and 'fill in the gaps (w/ automation, tools, expertise, whatever) and provide a map of what is needed to work on next
- Showing ROI and pipeline improvements along the way
- Risk-based testing approaches and how to build them into a pipeline
- Various aspects of a pipeline that can be overlooked as we might focus too much on just putting code into it and it plopping out the other side into production.

The success of an engineering organization and a business reliant on good software applications rises or falls to a significant degree with how solid the quality software pipeline is and how it is used. If great ideas and awesome features are created without the necessary quality processes to ensure they work, are released in a timely and reliable fashion, and continue to work well for the end users then users will go elsewhere. This can happen either immediately or over time and every other aspect of the business will suffer. However, with the right attention and consistent progress toward smart high quality pipelines this will positively impact many aspects of the business. I hope that some ideas presented here will help you and your business ensure quality with quick releases full of great features and changes in today's modern software world!

References

Book:

Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy. 2016 *Google Site Reliability Engineering.* O'Reilly Media, Incorporated.

Atul Gawande. 2011. *The Checklist Manifesto.* New York, NY: Metropolitan Books